Approximate Caching for Continuous Queries over Distributed Data Sources[*]

Chris Olston and Jennifer Widom
Stanford University
{olston, widom}@cs.stanford.edu

Abstract

Monitoring continuous queries over distributed data sources typically requires replicating data continuously at a central location for query monitoring, incurring significant communication overhead when source data is updated. We propose a new technique that reduces communication overhead by using *approximate caching*. Our approach enables users to register continuous queries with *precision constraints*. The system caches approximate data values with just enough accuracy to meet the precision constraints of all registered continuous queries at all times, while dynamically and adaptively allocating imprecision among cached values using an algorithm that minimizes data refreshes. Through experimental simulation over synthetic and real-world data, we demonstrate the effectiveness of our approach in reducing communication costs significantly compared with other approaches. Most importantly, we show that our algorithm enables users to trade precision for communication cost at a fine granularity by individually adjusting the precision constraints of continuous queries in a large multi-query workload, a feature that no known previous algorithm can provide.

1 Introduction

Continuous queries are used to monitor data as it changes over time, *e.g.*, [BW01, CDTW00, LPT99]. A system for processing continuous queries (CQ's) is expected to supply current query answers to users at all times, or alternatively be prepared to provide current answers on demand, without significant delay. To process CQ's that span distributed data sources, a system typically replicates data continuously at a central location for query monitoring [CDTW00], incurring significant communication overhead. We offer an effective method for reducing communication costs, taking advantage of the fact that many applications do not require exact consistency for their continuous queries—examples are discussed below. Instead of storing *stale* (out of date) copies of remote source data, our system stores *approximate* copies [DKP$^+$01, OLW01, OW00, YV00a]. Cached approximate values provide guaranteed bounds on the accuracy of continuous query results at any point in time. (Such guarantees cannot be provided by stale copies except under special conditions where data change rates are highly predictable.) Furthermore, through quantitative *precision constraints* supplied with continuous queries, users are offered fine-grained control over the tradeoff between precision and communication performance. Our system is called TRAPP/CQ, for *Tradeoff in Replication Precision and Performance for Continuous Queries*. (In Section 2 we place the contribution of this paper in the context of our overall TRAPP project.)

Many continuous query applications do not need exact answers, yet require quantitative guarantees regarding the precision of approximate answers [YV00b]. For example, managing complex computer networks requires tools that, among other things, continually report the status of network elements in real time, *e.g.*, [DR01, vRB01]. Network monitoring applications do not typically require exact answers [vRB01]. Thus, our approach can be used to reduce monitoring overhead by maintaining approximate synchronization between distributed network elements and a central monitoring station, while still providing quantitative precision guarantees for the approximate answers reported. As a second example, consider wireless sensor networks, *e.g.*, [EGPS01, KKP99, MF02, PK00], which enable continuous monitoring of environmental conditions such as light, temperature, sound, vibration, structural strain, etc. [MHSR02]. Since the battery life of miniature sensors is severely limited, and radio usage is the dominant factor determining battery life [MBC$^+$01, PK00], it is crucial to reduce the amount of data transmitted, even if a small increase in local processing by the sensor is required [MF02]. Many applications that rely on sensor data can tolerate ap-

[*]This work was supported by the National Science Foundation under grants IIS-9817799 and IIS-0118173 and by a National Science Foundation graduate research fellowship.

proximate answers having a controlled degree of imprecision [MBC$^+$01], making our approach ideal for reducing data transmission. Other examples with continuous queries over distributed data that can tolerate a bounded amount of imprecision include stock quote services, online auctions, wide-area resource accounting, and load balancing for replicated servers [YV00b].

1.1 Motivation and Overview of Approach

We focus on continuous queries that maintain aggregate values over numeric (real) data objects that may be distributed across many data sources. Aggregation is common in continuous query environments, including in wireless sensor deployments, where individual sensor readings are often combined [MF02], and in network monitoring, where it is often necessary to aggregate measurements from distributed network elements [DR01, vRB01]. The conventional answer to an aggregation query is a single real value. In TRAPP/CQ, we define a *bounded approximate answer* (hereafter called *bounded answer*) to be a pair of real values L and H that define an interval $[L, H]$ in which the precise answer is guaranteed to lie. Precision is quantified as the width of the range $(H - L)$, with 0 corresponding to exact precision and ∞ representing unbounded imprecision. A *precision constraint* for a continuous query is a user-specified constant $\delta \geq 0$ denoting a maximum acceptable interval width for the answer, *i.e.*, $0 \leq H - L \leq \delta$ at all times.

To provide guaranteed bounds $[L, H]$ as answers to continuous queries at all times, TRAPP/CQ requires cooperation between data sources and a centralized cache where queries are evaluated. Specifically, when a source refreshes the cache's value for a data object O, rather than sending the exact value of O, the source sends a *bound* $[L_O, H_O]$. The source guarantees that the actual value of O will stay in this bound, or if the value does exceed the bound then the source will immediately send a new refresh.[1] Thus, the cache stores the bound $[L_O, H_O]$ for each data object O instead of an exact value, and the cache can be assured that the source (master) value of O is within the bound. (We assume that the time to refresh a bound is small enough that the imprecision introduced is insignificant.) Continuous queries are registered at the cache, and the cached bounds are used to provide a continuous answer (or answer on demand at any time), also expressed in terms of a bound.

Each continuous query Q registered with the system has an associated precision constraint δ_Q. We assume any number of arbitrary CQ's with arbitrary individual precision constraints. The challenge is to ensure that at all times the bounded answer to every continuous query Q is of adequate precision, *i.e.*, has width at most δ_Q, while minimizing total communication cost. As a simple example, consider a single CQ requesting the average of n data values at different sources, with a precision constraint δ. We can show arithmetically that the width of the answer bound is the average of the widths of the n individual input bounds. Thus, one obvious way to guarantee the precision constraint is to maintain a bound of width δ on each of the n objects. Although this simple policy, which we call *uniform allocation*, is correct (the answer bound is guaranteed to satisfy the precision constraint at all times), it is not generally the best policy. To see why, it is important to understand the effects of cached value bound width [OLW01]. A cached bound that is narrow, *i.e.*, $H - L$ is small, enables continuous queries to maintain more precise answers, but is likely to incur more frequent refreshes. Conversely, a cached bound that that is wide, *i.e.*, $H - L$ is large, requires fewer refreshes but causes more imprecision in query answers. Uniform allocation can perform poorly for the following two reasons:

1. If multiple continuous queries are issued on overlapping sets of objects, different bound widths may be assigned to the same object. While we could simply choose to maintain the narrowest bound, the higher refresh cost may be wasted on all but a few queries.

2. Uniform bound allocation does not account for data values that change at different rates. In this case, we prefer to allocate wider bounds to data values that change rapidly, and narrower bounds to the rest.

Our performance experiments that compare uniform against nonuniform bound allocation policies provide strong empirical confirmation of these observations.

Reason 2 above indicates that a good nonuniform bound allocation policy depends heavily on the data change rates, which are likely to vary over time, especially during the long lifespan of continuous queries [MHSR02]. Therefore, in addition to nonuniformity, we propose an *adaptive* policy, in which bound widths are adjusted continually to match current conditions. Determining the best bound width allocation at each point in time without incurring excessive communication overhead is challenging, since it would seem to require a single site to have continual knowledge of data change rates across potentially hundreds of distributed sources. Moreover, the problem is complicated by reason 1 above: we may have many continuous queries with different precision constraints involving overlapping sets of data objects.

We have developed a low-overhead algorithm for setting bound widths adaptively to reduce communication costs while always guaranteeing to meet the precision constraints

[1]Note that our algorithm does require some state and computation at the sources. It is widely predicted that next-generation network monitoring devices [DR01, FDL$^+$01] and "smart" wireless sensors [EGPS01, KKP99] will have on-board computation capability. Since each distributed sensor or network monitor typically maintains at most a few values, only a small amount of state and computation is required.

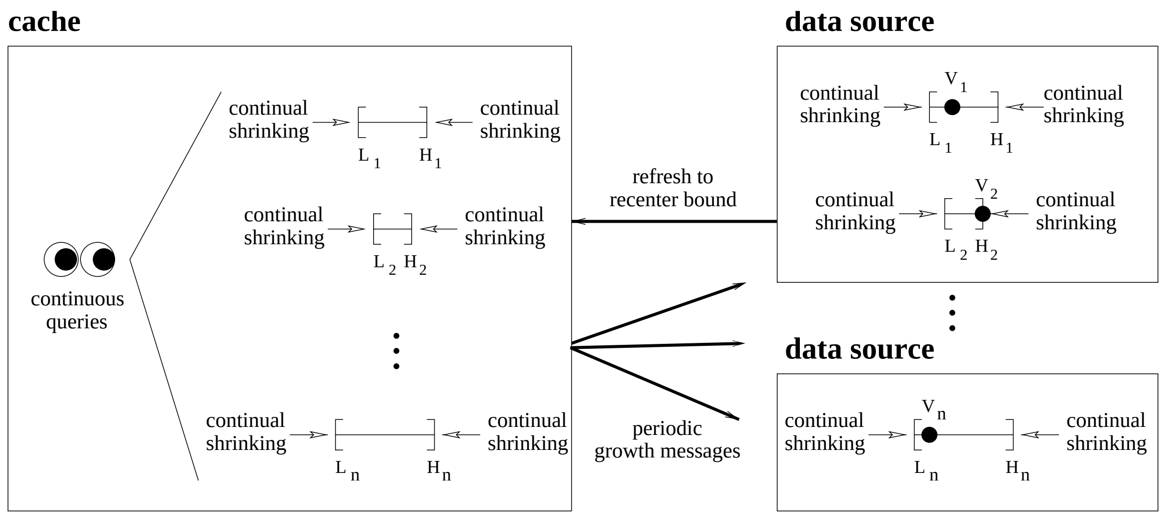

Figure 1: The TRAPP/CQ approach to approximate caching for continuous queries.

of an arbitrary set of registered CQ's. The basic idea is as follows. The cache shrinks the bound on each of its approximate data values continually over time, at a predefined rate. The sources are aware of the shrink rate, so they can maintain a mirror of the cached bounds and perform a refresh whenever a bound no longer contains its corresponding exact value (either because the object was updated or the bound shrank too far). Assuming the bounds begin in a state where all CQ precision constraints are satisfied (we will guarantee this to be the case), shrinking bounds only improves precision, so no precision constraint can become violated. Periodically, when the bounds have all shrunk by a certain degree, the cache takes the "leftover" width and reallocates it to the objects it benefits the most, ensuring all precision constraints remain satisfied. The cache then notifies the corresponding sources of the bound width increases.

The approach is illustrated in Figure 1. On the right we see the sources, each of which stores the master value for one or more objects, along with a bound for each object that shrinks continually over time. Whenever a source detects that an object's bound no longer contains its exact value, it recenters the bound around the current value, using the same bound width, and notifies the cache via a refresh message. The cache, shown on the left, maintains a continually shrinking bound for each object that mirrors the source bound at all times. The fact that bound shrinking is performed simultaneously at both the cache and the sources without explicit coordination is one key to achieving low communication overhead. An object's bound is only resized explicitly on occasion: the cache periodically selects certain objects to increase their bounds and notifies the corresponding sources via growth messages, as shown in Figure 1. The cache uses selective growth to tune the width allocation adaptively, the second crucial component to achieving low communication overhead. The goal is to guarantee individual precision constraints over arbitrary overlapping CQ's, without using excessively narrow cached approximations that would force unnecessary refreshes.

We provide a detailed description and mathematical justification of our bound width adjustment algorithm based on continual shrinkage and periodic selective growth in Sections 3 and 4. Then, in Section 5, we provide experimental evidence that our approach significantly reduces overall communication cost compared to a uniform allocation policy, on both synthetic and real data. Moreover, we demonstrate that even in the presence of a large number of overlapping continuous queries, TRAPP/CQ enables a smooth tradeoff between precision and communication cost.

2 Related Work

Our previous work in the TRAPP project [OLW01, OW00] established the idea of using cached approximations and queries with precision constraints to offer a smooth tradeoff between precision and performance in data caching environments. However, that work addressed *one-time* rather than continuous queries, resulting in a very different approach. Specifically, [OW00] developed algorithms for optimally combining approximate cached data with exact source data to meet the precision requirement for a single query at a single time. Follow-on work [OLW01] proposed a technique for adjusting cached approximations to minimize the overall communication cost under a workload of one-time queries like those in [OW00]. The approach in [OLW01] exploits the property that for many one-time queries, answer precision can be improved to acceptable levels by accessing remote sources at query-time. In this paper we focus on applications that require continuous answers to queries, and cannot tolerate the delay imposed by accessing remote sources when the answer to a query is requested. We propose a new technique for tuning the precision of cached approximations automatically and adaptively, minimizing communication cost while still maintaining the ability to answer continuous queries directly from the cache with adequate precision at all times.

In *quasi copies* [ABGMA88], cached approximations are permitted to deviate from exact source values by constrained amounts, thereby bounding the level of imprecision. However, this work does not address queries over multiple data values, whose answer precision is a function of the precision of the input values. We focus on aggregation queries and provide an algorithm for adjusting the relative precision of different cached approximations to minimize the cost of meeting precision requirements for any set of aggregation queries.

Recent work on reactive network monitoring [DR01] addresses scenarios where users wish to be notified whenever the sum of a set of values from distributed sources exceeds a prespecified critical value. In their solution each source notifies a central cache whenever its value exceeds a certain threshold, which can be either a fixed constant or a value that increases linearly over time. The local thresholds are set to guarantee that in the absence of notifications, the cache knows that the sum of the source values is less than the critical value. The thresholds in [DR01], which are related to the bounds in our algorithm, are set uniformly across all sources. Similarly, in [YV00b], which focuses on bounded approximate values under symmetric replication, error bounds are allocated uniformly across all sites that can perform updates. In contrast to these approaches, we propose a technique in which bound widths are allocated nonuniformly and adjusted adaptively based on refresh costs and change rates, in order to reduce communication cost as much as possible.

Finally, in the *demarcation protocol* [BGM92], sources communicate among themselves to verify numerical consistency constraints across sources. This approach could in principle be applied to our setting: sources could renegotiate bound widths in a peer-to-peer manner with the goal of reducing the frequency of cache refreshes. In many scenarios it may be impractical for sources to keep track of the other sources involved in a continuous query (or many continuous queries) and communicate with them directly, and even if practical it may be necessary to contact multiple peers before finding one with adequate "spare" bound width to share. It seems unlikely that the overhead of inter-source communication is warranted to potentially save a single cache refresh. Furthermore, the bound width adjustment algorithm in the demarcation protocol is not designed for the purpose of minimizing cache refreshes, and it does not accommodate multiple queries with overlapping query sets.

3 Algorithm Description

In this section we describe our algorithm for adjusting bound widths adaptively. Recall that our goal is to minimize communication costs while satisfying the precision constraints of all queries at all times. We consider continuous queries that operate over any fixed subset of the cached data values. (We do not consider selection predicates over approximately cached values, and we assume that all insertions and deletions are propagated immediately to the cache.) Queries can perform any of the five standard relational aggregation functions: COUNT, MIN, MAX, SUM, and AVG. Of these, COUNT can always be computed exactly in our setting, computing AVG reduces to computing SUM once COUNT is known, and MIN and MAX are symmetric. Therefore, from this point forward we discuss primarily the SUM and MIN functions. (Note that queries can request the value of an individual data object by posing a SUM query over a single object.) For flexibility we also allow objects to be weighted in SUM and AVG queries, formalized in Section 3.1.

Each registered continuous query Q_j specifies a *query set* $\mathcal{S}_j$ of objects and a *precision constraint* δ_j. The query set $\mathcal{S}_j$ is a subset of a set of n cached data objects $O_1, O_2, \ldots, O_n$. Each data object O_i has an exact value V_i stored at a remote source and a bound $[L_i, H_i]$ of width $W_i = H_i - L_i$ stored at the cache and mirrored at its source. We must ensure:

1. For each object O_i, $L_i \le V_i \le H_i$ at all times.
2. For each continuous query Q_j, the current answer $[L, H]$ to Q_j computed from the cached bounds for the objects in $\mathcal{S}_j$ satisfies Q_j's precision constraint, *i.e.*, $H - L \le \delta_j$ at all times.

As described earlier, to ensure condition 1, the source is responsible for noticing if the current value V_i becomes outside of the current bound (see [OW00] for some discussion of efficient detection mechanisms), and sending a refresh if so. During a refresh, the source sends a new bound to the cache that has the same width as the old one but is recentered around the current exact value V_i, so that $V_i - L_i = H_i - V_i$. Bounds shrink continually by default, so condition 2 can be violated only when the cache explicitly grows certain bounds. Our bound growth algorithm described below ensures that condition 2 remains valid at all times.

When the cache sends a bound growth message for object O_i to its source, or the source sends a refresh message to the cache, we model the cost as a known numerical constant C_i. (The possibility of batching growth messages or refreshes for multiple objects at the same source is a topic of future work.) The goal in setting and adjusting bound widths is to minimize the total cost incurred while satisfying all precision constraints at all times. In the following

Symbol	*Meaning*
n	number of cached data objects
O_i	data object ($i = 1 \ldots n$)
V_i	exact source value of object O_i
$[L_i, H_i]$	cached bound for object O_i
W_i	width of bound for object O_i ($W_i = H_i - L_i$)
U_i	pending width of bound for object O_i
C_i	cache $\leftrightarrow$ source communication cost for O_i
$\mathcal{C}$	overall communication cost
m	number of registered continuous queries
Q_j	registered query ($j = 1 \ldots m$)
$\mathcal{S}_j$	set of objects queried by Q_j
$K_{i,j}$	weight of object O_i in query Q_j
δ_j	precision constraint of query Q_j
$\mathcal{T}$	adjustment period (algorithm parameter)
S	shrink percentage (algorithm parameter)
A	acceleration constant (algorithm parameter)
N	refresh window size (algorithm parameter)
P_i	refresh period of O_i (based on last N refreshes)
B_i	burden score of O_i (computed every $\mathcal{T}$ seconds)
T_j	burden target of Q_j (computed every $\mathcal{T}$ seconds)
D_i	deviation of O_i (determines growth priority)

Table 1: Model and algorithm symbols.

two subsections, we describe our approach for SUM and MIN queries, respectively.

For convenience, the symbols we have introduced and others we will introduce later in this section are summarized in Table 1.

3.1 SUM Queries

We first consider continuous queries that monitor the weighted sum (or weighted average by extension) of a set of data values. Say there are m such registered continuous queries $Q_1, Q_2, \ldots, Q_m$. For each query Q_j, each object O_i in Q_j's query set $\mathcal{S}_j$ may have an associated (positive or negative) weight $K_{i,j}$ for the query. We let $K_{i,j} = 1$ in the absence of a specified weight. For notational convenience let us say $K_{k,j} = 0$ for all objects O_k in the cache but not in Q_j's query set. Then the exact answer to SUM query Q_j is $\sum_{1 \le i \le n} K_{i,j} \cdot V_i$, and our goal is to be able to continuously compute an approximate answer from cached bounds that is within Q_j's precision constraint δ_j. Note that weighted SUM queries are quite flexible, *e.g.*, we can monitor the difference between two values by using weights 1 and -1.

Before presenting our general adaptive algorithm for adjusting bound widths, we describe two simple cases in which the bound width of certain objects should remain fixed. First, consider an object O that is involved only in queries that simply request a bound on the value of O alone (SUM queries over one value). Then it suffices to fix the bound width of O to be the smallest of the precision constraints: $W_i = \min(\delta_j)$ for queries Q_j with $\mathcal{S}_j = \{O\}$. Second, for objects that are not included in any currently registered query, the cached bound should be fixed at $[-\infty, \infty]$ so that the value is never refreshed. The remainder of the objects, namely those that are involved in at least one query over multiple objects, pose our real challenge.

To guarantee that all precision constraints are met, the following constraint must hold for each query Q_j (see Appendix A for a derivation):

$$\sum_{1 \le i \le n} |K_{i,j}| \cdot W_i \le \delta_j$$

In other words, the weighted sum (in absolute value) of bound widths for each query must not exceed the precision constraint. Initially, the bounds can be set in any way that meets the precision constraint of every query, *e.g.*, using a simple greedy algorithm. Then, as discussed in Section 1.1, our general strategy is to reallocate bound width adaptively among the objects participating in each query. Reallocation is accomplished with low communication overhead by having bounds shrink continually over time and having the cache periodically select one or more bounds to grow based on current conditions. In Section 3.1.1 we describe the exact way in which bounds are shrunk in our algorithm, and then in Section 3.1.2 we describe when and how bounds are grown.

3.1.1 Bound Shrinking

Every object O_i has a corresponding *pending width* U_i that is maintained simultaneously at both the cache and source. Initially $U_i = W_i$. Once every second (or other time unit, but we will use seconds), if O_i is not refreshed during that second, then U_i is decreased by setting $U_i := U_i \cdot (1 - S) \cdot \alpha$, where S is a global parameter called the *shrink percentage* and α is the *acceleration factor*, which we describe in a moment. Ignoring the α term for now, the effect is to decrease the pending width by the fraction S every second. Periodically, every $\mathcal{T}$ seconds, the bound $[L_i, H_i]$ for O_i is shrunk symmetrically at both the source and the cache by setting $L_i = L_i + \frac{W_i - U_i}{2}$ and $H_i = H_i - \frac{W_i - U_i}{2}$. The constant $\mathcal{T}$ is a global parameter called the *adjustment period*. All adjustments to the bound width (decreases as well as increases) occur at intervals of $\mathcal{T}$ seconds. Note that refreshes can occur at any time but they simply reposition bounds without altering the width, as discussed earlier.

The acceleration factor is defined as $\alpha = \min\{\frac{\mathcal{T}}{A \cdot t_i}, 1.0\}$, where A is a global parameter called the *acceleration constant* and t_i is the time elapsed (in seconds) since the last refresh of O_i. The acceleration factor serves to accelerate the rate of shrinking if no refresh has occurred recently. The larger the value A, the more adaptive the algorithm. We will discuss good settings for A and other algorithm parameters in Section 5.1.

3.1.2 Bound Growing

Every $\mathcal{T}$ seconds, when all the bounds shrink automatically as described in the previous section, the cache selects certain bounds to grow instead. The first step in this process is to assign a numerical *burden score* B_i to each cached object O_i. Conceptually, the burden score embodies the degree to which an object is contributing to the overall communication cost. It is computed as $B_i = \frac{C_i}{P_i \cdot W_i}$, where recall that C_i is the cost to refresh object O_i, and W_i is the width of the cached bound. P_i is O_i's estimated *refresh period*, computed as the average time interval between O_i's N most recent refreshes, where N is a parameter called the *refresh window size*. The burden formula is fairly intuitive since, *e.g.*, a wide bound or long refresh period reduces B_i. The exact mathematical derivation is given in Section 4.

Once each object's refresh period and burden score have been computed, the second step is to assign a value T_j, called the *burden target*, to each weighted SUM query Q_j. Conceptually, the burden target of a query represents the lowest overall burden required of the objects in the query in order to meet the precision constraint at all times. Appendix B describes how burden targets are computed based on mathematical results derived later in Section 4. It turns out that assigning burden targets requires solving a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities. As described in Section 5.4, we use an iterative linear solver to assign burden targets efficiently.

Once a burden target has been assigned to each query, the third step is to compute for each object O_i its *deviation* D_i:

$$D_i = \max\{B_i - \sum_{1 \le j \le m} |K_{i,j}| \cdot T_j, 0\}$$

Deviation indicates the degree to which an object is "overburdened" with respect to the burden targets of the queries that access it. To achieve low overall communication cost, it is desirable to equally distribute the burden across all objects involved in a given query. We will justify this claim mathematically in Section 4.

To see how we can even out burden, recall that the burden score of object O_i is $B_i = \frac{C_i}{P_i \cdot W_i}$, so if the bound $[L_i, H_i]$ were to increase in size, B_i would decrease.[2] Therefore, the burden score of an overburdened object can be reduced by growing its bound. Growth is allocated to bounds using the following greedy strategy. Objects are considered in decreasing order of deviation, so that the most overburdened objects are considered first. (It is important that ties be resolved randomly to prevent objects having the same deviation—most notably 0—from repeatedly being considered in the same order.) When object O_i is considered, the maximum possible amount by which the bound can be grown without violating the precision constraint of any query is computed as:

$$\Delta W_i = \min_{1 \le j \le m | O_i \in \mathcal{S}_j} \frac{\delta_j - \sum_{1 \le k \le n} |K_{k,j}| \cdot W_k}{|K_{i,j}|}$$

If $\Delta W_i = 0$, then no action is taken. For each nonzero growth value, the cache increases the width of the bound for O_i symmetrically by setting $L_i = L_i - \frac{\Delta W_i}{2}$ and $H_i = H_i + \frac{\Delta W_i}{2}$. The cache immediately sends a message to object O_i's source to inform it of the new bound width, and the pending width U_i is reset to W_i, as discussed in Section 3.1.1.

In summary, the procedure for determining how much to grow each cached bound is as follows. First, each object is assigned a burden score based on its refresh cost, estimated refresh period, and current bound width. Second, each query is assigned a burden target by invoking an iterative linear solver. Each object is then assigned a deviation value based on the difference between its burden score and the burden targets of the queries that access it. Finally, the objects are considered in order of decreasing deviation, and each object O_i is assigned the maximum possible bound growth ΔW_i when it is considered.

Let us consider the complexity of our bound growth algorithm, which is executed once every $\mathcal{T}$ seconds. Most of the steps involve a simple computation per object, and the objects must be sorted once. In the last step, to compute ΔW_i efficiently, the cache can continually track the difference ("leftover width") between each query's precision constraint and the current answer's bound width. Then for each object we use the precomputed leftover width value for each query over that object. We expect the iterative linear solver used to compute burden targets to dominate the computation. In Section 5.4 we demonstrate the scalability of the solver, and we verify that it requires only a small fraction of the computing resources at the cache.

3.2 MIN Queries

We now consider MIN queries, and show that for the purposes of bound width setting they can be treated as a collection of SUM queries. Consider a MIN query Q_j over query set $\mathcal{S}_j$ with precision constraint δ_j. First, we show that if the bound for each object $O_i \in \mathcal{S}_j$ has width at most δ_j, the precision constraint is always met. To see this fact, observe that the answer $[L, H] = [\min(L_i), \min(H_i)]$, and it has width $H - L = \min(H_i) - \min(L_i) \le H_k - \min(L_i)$, for the upper bound H_k of any object $O_k \in \mathcal{S}_j$. If we choose O_k to be the object with the lowest lower bound, *i.e.*, $L_k = \min(L_i)$, we obtain $H - L \le H_k - L_k$. Thus, if all bounds have width at most δ_j, then the answer bound has width $H - L \le \delta_j$.

We now show the converse: if the bound for some $O_i \in \mathcal{S}_j$ has width greater than δ_j, then the precision constraint cannot be guaranteed. To see this fact, consider an

[2]This reasoning relies on P_i not decreasing when W_i increases, a fact that holds intuitively and is discussed further in Section 4.

object $O_i \in \mathcal{S}_j$ whose value is far greater than the minimum value of the queried objects. It may seem safe to assign a bound $[L_i, H_i]$ exceeding δ_j to O_i, as long as L_i is greater than the lowest lower bound, since $[L_i, H_i]$ will not contribute to the answer bound. However, if the cache receives a refresh of one or more objects, causing L_i to suddenly become the lowest lower bound and H_i the lowest upper bound, then the new answer bound has width greater than δ_j. Although this situation could be remedied by requesting a tighter bound for O_i from its source, this procedure would incur a delay during which Q_j's answer bound violates its precision constraint, breaking our requirement of continuous precision for continuous queries. Therefore, for a MIN query Q_j, the bound for each queried object must have width at most δ_j at all times, and those widths are guaranteed to uphold the precision constraint.

Based on the above observations, for the purposes of bound width setting, a MIN query Q_j with precision constraint δ_j over a set of objects $\mathcal{S}_j$ is equivalent to a set of single-object queries over each $O_i \in \mathcal{S}_j$ with precision constraint δ_j for each. Since single-object queries are SUM queries over one object with weight 1, the techniques in Section 3.1 can be applied to MIN queries without modification.

In summary, since AVG can be computed from SUM, and MAX is symmetric to MIN, the Section 3.1 techniques can be used for any workload consisting of a combination of SUM, AVG, MIN, and MAX queries.

4 Mathematical Justification for Bound Growth Strategy

In this section we use a mathematical model for the behavior of objects with cached bounds to justify the bound growth allocation strategy presented in Section 3.1.2. We model the behavior of objects with cached bounds as follows. For an object O_i whose exact value V_i varies with time, we assume that the refresh period P_i is a function of the bound width $W_i = H_i - L_i$, and signify this relationship by writing $P_i = P_i(W_i)$. Intuitively, when a bound is narrow, the actual value is likely to exceed it more often and therefore the refresh period will be short. Conversely, when a bound is wide we expect the refresh period to be longer. The precise relationship between W_i and P_i depends on the behavior of V_i.

Since each refresh of object O_i incurs a cost C_i, we can express the communication cost of the entire system as:

$$\mathcal{C} = \sum_{1 \le i \le n} \frac{C_i}{P_i(W_i)}$$

If no continuous queries are registered, then zero cost can be achieved by setting all bounds to $[-\infty, \infty]$. However, as derived in Appendix A, each query Q_j with weights $K_{1,j}, K_{2,j}, \ldots, K_{n,j}$ and precision constraint δ_j imposes the following constraint on the bound widths:

$$\sum_{1 \le i \le n} |K_{i,j}| \cdot W_i \le \delta_j$$

(Recall that we set $K_{i,j} = 0$ for objects $O_i \notin \mathcal{S}_j$, and that all queries can be treated as weighted SUM queries.) We are now faced with the optimization problem of minimizing the overall cost $\mathcal{C}$ while satisfying the above constraint for each of m queries $Q_1, Q_2, \ldots, Q_m$.

Unless the function $P_i(W_i)$ is inversely proportional to W_i, which is unlikely as discussed above, we are faced with a nonlinear optimization problem with inequality constraints. Since such problems are very difficult to solve, we decided to try treating the inequality constraints as equality constraints to get an idea of the form of the solution. We can apply the method of Lagrange Multipliers [Ste91] to minimize $\mathcal{C}$ under a set of m equality constraints of the form:

$$\sum_{1 \le i \le n} |K_{i,j}| \cdot W_i = \delta_j$$

The solution [Ste91] has the property that there are a set of m constants $\lambda_1, \lambda_2, \ldots, \lambda_m$ such that for all i:

$$C_i \cdot \frac{\partial}{\partial W_i}\left(\frac{1}{P_i(W_i)}\right) = \sum_{1 \le j \le m} |K_{i,j}| \cdot \lambda_j$$

To evaluate the derivative we make the assumption that the function $P_i(W_i)$ has roughly the form $P_i(W_i) = Z_i \cdot (W_i)^p$, where each Z_i is an arbitrary constant and p can be any positive integer. For example, this model with $p = 2$ applies to data that follows a random walk pattern, as shown in Appendix C. Assuming $P_i(W_i)$ roughly follows this form, we can evaluate the partial derivative to obtain the following expression:

$$\frac{C_i}{P_i \cdot W_i} = M \cdot \sum_{1 \le j \le m} |K_{i,j}| \cdot \lambda_j$$

where M is a constant. Finally, let the burden target $T_j = M \cdot \lambda_j$ and recall that the burden score $B_i = \frac{C_i}{P_i \cdot W_i}$, giving:

$$B_i = \frac{C_i}{P_i \cdot W_i} = \sum_{1 \le j \le m} |K_{i,j}| \cdot T_j$$

According to this formula, we want P_i and W_i to be set such that the burden score B_i of each object O_i roughly equals the weighted sum of the burden targets of all queries over O_i. Our algorithm described in Section 3 converges to this state by monitoring the burden scores and increasing W_i, and consequently P_i as well, to decrease B_i when it becomes significantly higher than the weighted sum of estimated targets.

5 Experimental Validation

We evaluated our technique with a discrete event simulator using one synthetic data set and two real-world data sets. For the synthetic data set, we simulated ten sources and generated data for one object per source following a random walk pattern, each with a randomly-assigned step size. We also gathered two real-world data sets, one from stock market data and the other from network traffic data. For the stock market data, we used the prices of 100 randomly-selected stock symbols as they varied during one month of trading (December 1995). For the network traffic data, we used publicly available traces of network traffic levels between hosts distributed over a wide area during a two hour period [PF95]. For each host, the data values we use represent a one-minute moving window average of network traffic every second, and we randomly selected 200 hosts as our simulated data sources.

In all of our experiments, we simulated both uniform costs and nonuniform costs. In the latter, the cost C_i to send a message between the cache and the source of each object O_i was assigned randomly to an integer between 1 and 10. The results for uniform and nonuniform costs were very similar in all cases, so due to space constraints we show results for uniform costs only. Query loads are introduced in each section.

5.1 Parameter Settings

The first step in our experimentation was to determine good settings for the four algorithm parameters $\mathcal{T}$ (adjustment period), S (shrink percentage), A (acceleration constant), and N (refresh window size). Due to space constraints we simply state our findings, and note that in all cases we did conclude that the algorithm is not highly sensitive to the exact parameter settings. We experimented with all of our data sets and with both uniform and nonuniform costs, and found that the following settings worked well in general: $\mathcal{T} = 10$, $S = 0.001$[3], $A = 0.01$, and $N = 2$.

5.2 Single Query

We now present our first experimental results showing the effectiveness of our algorithm. We begin by considering a simple case involving a single unweighted continuous AVG query over all objects. (In all of our experiments we use AVG instead of SUM to make the precision constraints more intuitive. Recall that the same algorithm is used for both.)

5.2.1 Comparison with Optimized Static Strategy

The goal of our first experiment is to show that our algorithm converges on the best possible bound widths, given a steady-state data set. For this purpose, we used random walk data, and we compared two unrealistic algorithms. In the "idealized" version of our algorithm, messages sent by the cache to sources instructing them to grow their bounds incur no communication cost. Instead, only refresh costs were measured, to focus on the bound width choices only. We compared the overall refresh cost against the refresh cost when bound widths are set statically using an optimization problem solver, described next.

The nature of random walk data makes it possible to simplify the problem of setting bound widths statically to a nonlinear optimization problem. Recall from Section 4 that the overall cost $\mathcal{C} = \sum_{1 \le i \le n} \frac{C_i}{P_i(W_i)}$, where $P_i(W_i)$ is the refresh period as a function of the bound width of O_i. In Appendix C, we derive an approximate formula for this function in the random walk case: $P_i(W_i) \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$, which depends on the step size s_i. If the step sizes of all the objects are known, then a good static bound width allocation can be found by solving the following nonlinear optimization problem: minimize $\sum_{1 \le i \le n} \frac{C_i \cdot (2 \cdot s_i)^2}{(W_i)^2}$ in the presence of m constraints of the form $\sum_{1 \le i \le n | O_i \in \mathcal{S}_j} W_i \le \delta_j$. (For now $m = 1$, but in Section 5.3.1 we experiment with the general case of larger m.) While nonlinear optimization problems with inequality constraints are difficult to solve exactly, an approximate solution can be obtained with methods that use iterative refinement. We used a package called FSQP [LZT97], iterating 1000 times with tight convergence requirements, to find static bound width settings as close as possible to optimal.

Figure 2 shows the results of comparing the idealized version of our adaptive algorithm against the optimized static allocation, under uniform costs. (As in all of our experiments, the results for nonuniform costs are similar.) The x-axis shows the precision constraint δ, and the y-axis shows the overall cost per second. These results demonstrate that our adaptive bound width setting algorithm converges on bounds that are on par with those selected statically by an optimizer based on knowledge of the random walk step sizes.

For reference, we also plot the cost of *exact caching*, in which all queried bounds have width zero and the cache maintains an exact copy of each source value involved in one or more queries. Clearly, the use of approximate caching offers the potential for enormous performance advantages over exact caching. The less stringent the precision constraint of the query, the lower the cost of maintaining the query answer at the cache. As we will see, subsequent experiments on more complex scenarios demonstrate similar results.

5.2.2 Evaluation on Real-World Data

In our next experiment, to evaluate the effectiveness of our algorithm in a realistic environment, we included all com-

[3] Setting $S = 0.001$ corresponds to a shrinkage rate of roughly 1% per second when $\mathcal{T} = 10$.

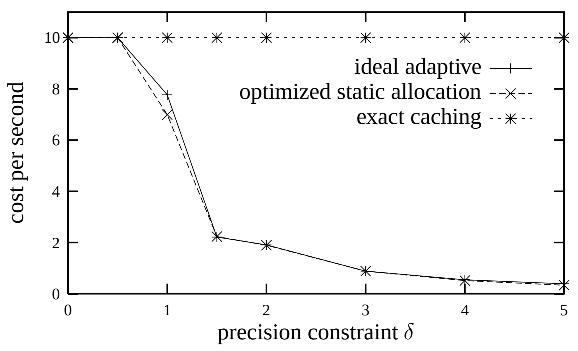

Figure 2: Ideal adaptive algorithm vs. optimized static allocation, random walk data.

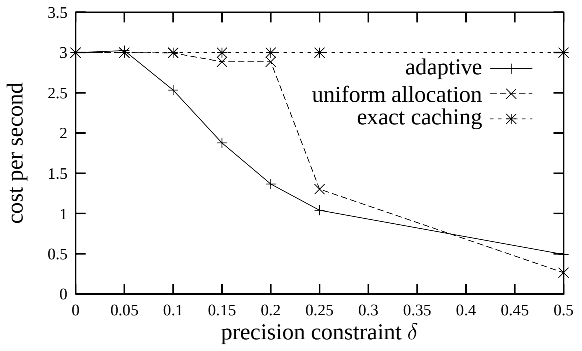

Figure 3: Adaptive algorithm vs. static allocation, stock market data.

munication messages in our measured cost and used real-world data. Since the optimized static bound width allocation described in Section 5.2.1 relies on knowing the random walk step size, it is not applicable to real-world data. Assuming data update patterns are not known in advance, the only obvious method of static allocation is to set all bound widths uniformly.

The graphs in Figures 3 and 4 compare the overall cost incurred by our adaptive algorithm compared to uniform static allocation for stock market and network traffic data, respectively. For the stock market data (Figure 3), the continuous query monitors the average stock price with precision constraint δ ranging from 0 to 0.5 dollars per share. For the network traffic data, the continuous query monitors the average traffic level with precision constraint δ ranging from 0 to 10×10^3 packets per second. Our algorithm significantly outperforms uniform static allocation for queries that can tolerate a moderate level of imprecision (small to medium precision constraints). For queries with very weak precision requirements (large precision constraints), even naive allocation schemes achieve low cost, and the slight additional overhead of our algorithm causes it to perform about on par with uniform static allocation.

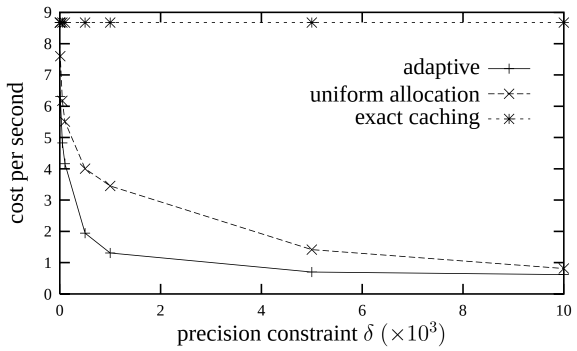

Figure 4: Adaptive algorithm vs. static allocation, network traffic data.

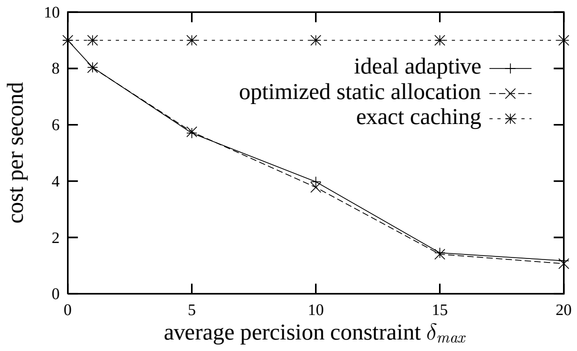

Figure 5: Ideal adaptive algorithm vs. optimized static allocation, multiple queries.

5.3 Multiple Queries

We now describe our experiments with multiple continuous queries having overlapping query sets.

5.3.1 Comparison with Optimized Static Strategy

Again using our random walk data set, we compared the idealized version of our algorithm (in which bound growth messages incur no cost) against the optimized static allocation, as in Section 5.2.1. This time we used a workload with five unweighted AVG queries whose query sets were chosen randomly from the 10 objects. The size of the query sets was assigned randomly between 2 and 5, and the precision constraint of each query was randomly assigned a value between 0 and δ_{max}, plotted on the x-axis. Figure 5 shows the result of these experiments. Again we see that our algorithm converges adaptively on bound widths that perform as well as those chosen statically by an optimizer with knowledge of random walk step sizes.

5.3.2 Evaluation on Real-World Data

Next we measured multiple queries over real-world data. As discussed in Section 1.1, uniform static bound allocation is inappropriate for multiple overlapping queries. Thus

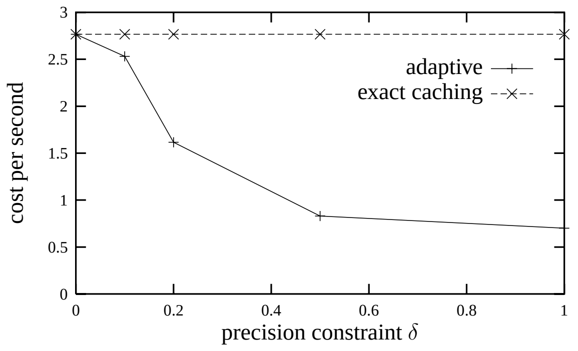

Figure 6: Adaptive algorithm vs. exact caching, multiple queries, stock data.

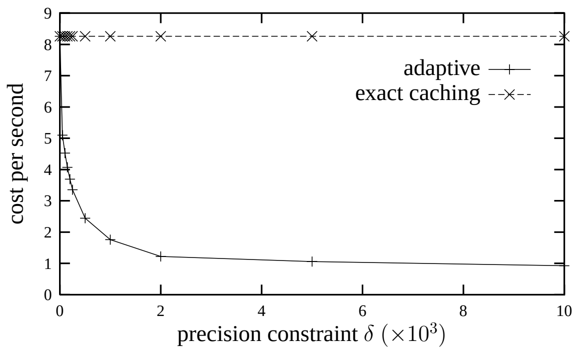

Figure 7: Adaptive algorithm vs. exact caching, multiple queries, network data.

there is no obvious point of comparison other than exact caching.

We used two different multiple query workloads in these experiments. First, we measured a workload with five unweighted AVG queries, each over a randomly-chosen half of the data objects (50 stock prices or 100 network traffic levels). Figures 6 and 7 show the results of our experiments on this five-query workload for stock market and network traffic data, respectively. As with all previous results reported, the overall cost decreases rapidly as the precision constraint is relaxed, offering significant performance gains over exact caching.

As a final experiment, we measured a mixed workload of 100 queries, of which 25 were queries over an individual object, 50 were AVG queries with random weights in $[-1, 1]$ over a randomly-selected 25% of the objects, and the remaining 25 were MIN queries over a randomly-selected 25% of the objects. For this workload, we lengthened the adjustment period $\mathcal{T}$ to 100 to reduce communication overhead. Figures 8 and 9 show the results for stock market and network traffic data, respectively. These graphs demonstrate that our algorithm is able to handle a large query mix, with similar results as smaller uniform workloads.

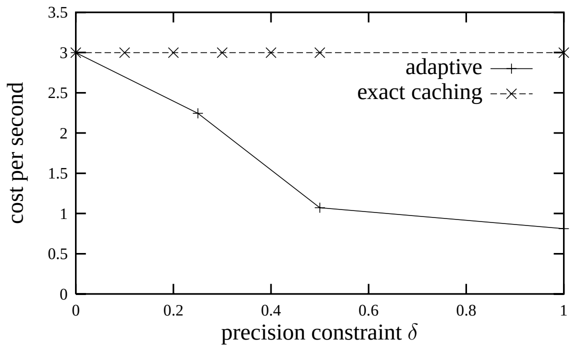

Figure 8: Adaptive algorithm vs. exact caching, mixed workload, stock data.

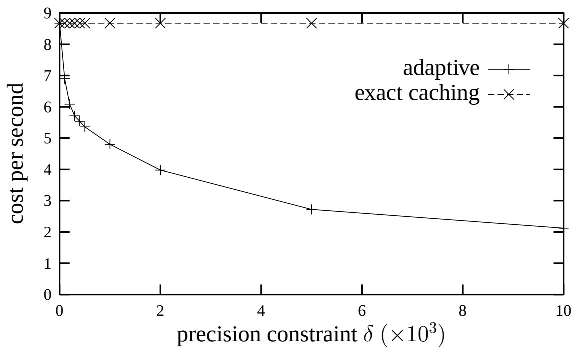

Figure 9: Adaptive algorithm vs. exact caching, mixed workload, network data.

5.4 Scalability

The critical component of our algorithm in terms of scalability is the linear system solver used to compute query burden targets (Section 3.1.2 and Appendix B). Recall that assigning burden targets requires solving a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities. Because solving this system of equations exactly at run-time is likely to be very expensive, we find an approximate solution by representing the equations as an m by $(m + 1)$ matrix and running an iterative solver until it converges within a small error ϵ. We use a publicly-available iterative solver package called *LASPack* [Ska96], although many alternatives exist. Note that the matrix tends to be quite sparse, since whenever the query sets $\mathcal{S}_x$ and $\mathcal{S}_y$ of two queries Q_x and Q_y are disjoint, the corresponding matrix entry is 0. For this reason, along with the fact that we can tune the number of iterations, burden target computation using an iterative linear system solver should scale well.

To test the scalability of our algorithm to a large number of queries, we generated two sets of workloads consisting of AVG queries over the network traffic data (results for stock market data were nearly identical). In one set of workloads, each query is over a randomly-selected 5% (10) of the data objects. In the second set of workloads,

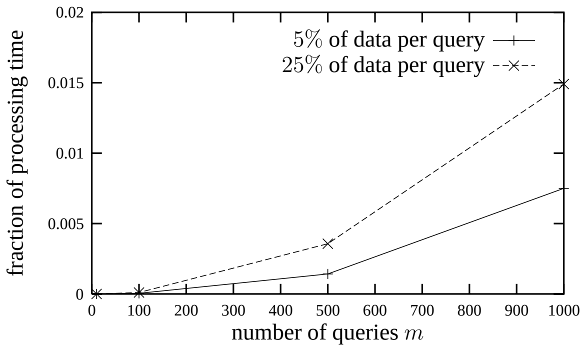

Figure 10: Scalability of linear system solver.

each query is over 25% (50) of the data objects, resulting in a much higher degree of overlap among queries. Varying the number of queries m, we measured the average running time on a Linux workstation with a 700 MHz Pentium III processor. We set the error tolerance for the LASPack iterative solver small enough that no change in the effectiveness of our overall algorithm could be detected. Figure 10 shows the fraction of available processing time used by the linear solver when it is invoked once every 100 seconds (when $\mathcal{T} = 100$). Allocating bound growth to handle 1000 queries over 25% of the data requires only around 1.5% of the processor time at the cache.

6 Summary

We specified and mathematically justified an adaptive algorithm used in TRAPP/CQ to achieve low communication cost under dynamically changing conditions, given a workload consisting of a large number of continuous queries with precision constraints over distributed source data. Our algorithm keeps the precision of all queries within permitted levels at all times, finding a low-cost solution by continually adjusting the relative allocation of imprecision among approximately cached objects: cached precision bounds shrink by default, and the cache periodically selects certain bounds to grow instead. TRAPP/CQ enables users or applications to trade precision for lower communication cost at a fine granularity by individually adjusting the precision constraints of continuous queries. This feature contrasts with traditional approaches, which either provide exact precision at high cost or unbounded imprecision at low cost.

To validate our technique, we performed a number of experiments over both synthetic and real-world data. We demonstrated that for a steady-state scenario our algorithm converges on bound widths that perform on par with those selected statically using an optimization problem solver with knowledge of data update behavior. We also showed that in the case of a single continuous query, our algorithm significantly outperforms a rival technique in which bound widths are allocated uniformly. Finally, we tested our algorithm with multiple continuous queries, a scenario in which uniform allocation does not apply, and demonstrated that our algorithm is consistently effective and offers a smooth tradeoff between precision and low communication cost.

Acknowledgments

We thank Hector Garcia-Molina and Cheng Yang for their helpful discussions and feedback.

References

[ABGMA88] R. Alonso, D. Barbara, H. Garcia-Molina, and S. Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 443–468, Venice, Italy, March 1988.

[BGM92] D. Barbara and H. Garcia-Molina. The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 373–387, Vienna, Austria, March 1992.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, September 2001.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 379–390, Dallas, Texas, May 2000.

[DKP$^+$01] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic Web data. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, China, May 2001.

[DR01] M. Dilman and D. Raz. Efficient reactive monitoring. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, Anchorage, Alaska, April 2001.

[EGPS01] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, Salt Lake City, Utah, May 2001.

[FDL$^+$01] C. Fraleigh, C. Diot, B. Lyles, S. Moon, P. Owezarski, D. Papagiannaki, and F. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Proceedings of the Workshop on Passive and Active Measurement*, Amsterdam, The Netherlands, April 2001.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, Massachusetts, 1989.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Network Monitoring (MobiCom 99)*, pages 271–278, Seattle, Washington, August 1999.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 11(4):610–628, 1999.

[LZT97] C. T. Lawrence, J. L. Zhou, and A. L. Tits. User's guide for CFSQP version 2.5: A C code for solving (large scale) constrained nonlinear (minimax) optimization problems, generating iterates satisfying all inequality constraints. Technical report TR-94-16r1, Institute for Systems Research, University of Maryland, 1997.

[MBC$^+$01] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. Low-power wireless sensor networks. In *Proceedings of the Fourteenth International Conference on VLSI Design*, Bangalore, India, January 2001.

[MF02] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, California, February 2002.

[MHSR02] S. Madden, J. M. Hellerstein, M. Shah, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (to appear)*, Madison, Wisconsin, June 2002.

[OLW01] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 355–366, Santa Barbara, California, May 2001.

[OW00] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 144–155, Cairo, Egypt, September 2000. (Extended version available at http://www-db.stanford.edu/pub/papers/trapp-ag.ps.).

[PF95] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[PK00] G.J. Pottie and W.J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):551–558, May 2000.

[Ska96] T. Skalicky. Laspack reference manual, 1996. http://www.tu-dresden.de/mwism/skalicky/laspack/laspack.html.

[Ste91] J. Stewart. *Calculus: Early Transcendentals, Second Edition*. Brooks/Cole, 1991.

[vRB01] R. van Renesse and K. Birman. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. Technical report, Cornell University, 2001.

[YV00a] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, October 2000.

[YV00b] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 123–133, Cairo, Egypt, September 2000.

A Derivation of Constraint Formula

We now derive the constraint formula given in Section 4. The bound $[L, H]$ on the answer to a weighted sum query Q_j is computed from the weight $K_{i,j}$ associated with each object O_i along with its cached bound $[L_i, H_i]$. The lowest possible weighted sum L occurs when the values of objects with a positive weight are as small as possible and the values of objects with a negative weight are as large as possible. In other words, whenever $K_{i,j} \geq 0$, the sum lower bound has $V_i = L_i$, and whenever $K_{i,j} < 0$, the sum lower bound has $V_i = H_i$. The converse holds for the weighted sum upper bound H. Therefore, a tight bound on the weighted sum answer is:

$$[L, H] = \left[\left(\sum_{1 \leq i \leq n | K_{i,j} \geq 0} K_{i,j} \cdot L_i + \sum_{1 \leq i \leq n | K_{i,j} < 0} K_{i,j} \cdot H_i \right), \left(\sum_{1 \leq i \leq n | K_{i,j} \geq 0} K_{i,j} \cdot H_i + \sum_{1 \leq i \leq n | K_{i,j} < 0} K_{i,j} \cdot L_i \right) \right]$$

which can be rewritten as:

$$[L, H] = \left[\left(\sum_{1 \leq i \leq n | K_{i,j} \geq 0} |K_{i,j}| \cdot L_i - \sum_{1 \leq i \leq n | K_{i,j} < 0} |K_{i,j}| \cdot H_i \right), \left(\sum_{1 \leq i \leq n | K_{i,j} \geq 0} |K_{i,j}| \cdot H_i - \sum_{1 \leq i \leq n | K_{i,j} < 0} |K_{i,j}| \cdot L_i \right) \right]$$

The answer bound width is $H - L$, which using the second formula above simplifies to $\sum_{1 \leq i \leq n} |K_{i,j}| \cdot (H_i - L_i)$.

B Burden Target Computation

We describe how to compute the burden target T_j for each query Q_j, given the burden score B_i of each object O_i. Recall from Section 3.1.2 that conceptually the burden target for a query represents the lowest overall burden required of the objects in the query in order to meet the precision constraint at all times. For motivation consider first the special case involving a single unweighted sum query Q_k over every object $O_1, \ldots, O_n$ with all weights $K_{i,k} = 1$. In this scenario, the goal for adjusting the burden scores simplifies to that of equalizing them (Section 4), so $B_1 = B_2 = \cdots = B_n = T_k$. Therefore, given a set of burden scores that may not be equal, a simple way to guess at an appropriate burden target T_k is to take the average of the current burden scores, *i.e.*, $T_k = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} B_i$. In this way, objects having higher than average burden scores will be given high priority for growth to lower their burden scores, and those having lower than average burden scores will shrink by default, thereby raising their burden scores. On subsequent iterations, the burden target T_k will be adjusted to be the new average burden score. This overall process results in convergence of the burden scores.

We now generalize to the case of multiple weighted queries over different sets of objects. Let $\theta_{i,j}$ represent the portion of object O_i's burden score corresponding to query Q_j so that $\sum_{1 \leq k \leq m} \theta_{i,k} = B_i$. For each object/query pair O_i/Q_j, we can express $\theta_{i,j}$ in terms of B_i, which is known, and the θ values for the other queries over O_i, which are unknown: $\theta_{i,j} = B_i - \sum_{1 \leq k \leq m, k \neq j} \theta_{i,k}$. Taking the average $\theta_{*,j}$ value for query Q_j across all queried objects $O_i \in \mathcal{S}_j$, weighted appropriately, we have $T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n | O_i \in \mathcal{S}_j} \frac{\theta_{i,j}}{|K_{i,j}|}$. If we substitute our expression for $\theta_{i,j}$ and replace each occurrence of $\theta_{i,k}$ by $|K_{i,k}| \cdot T_k$ for all $k \neq j$, we arrive at the following expression:

$$T_j = \frac{1}{|\mathcal{S}_j|} \cdot \sum_{1 \leq i \leq n | O_i \in \mathcal{S}_j} \left(\frac{B_i}{|K_{i,j}|} - \sum_{1 \leq k \leq m, k \neq j} \frac{|K_{i,k}|}{|K_{i,j}|} \cdot T_k \right)$$

This result is a system of m equations with $T_1, T_2, \ldots, T_m$ as m unknown quantities, which can be solved using a linear solver package.

C Refresh Period for Random Walk Data

We derive an expression for the expected refresh period $P_i(W_i)$ as a function of the bound width W_i for an object O_i that changes according to a random walk pattern. In the random walk model, after t steps of size s_i, the probability distribution of the value is a binomial distribution with variance $(s_i)^2 \cdot t$ [GKP89]. Chebyshev's Inequality [GKP89] gives an upper bound on the probability $\mathcal{P}$ that the value is beyond any distance k from the starting point: $\mathcal{P} \leq t \cdot (\frac{s_i}{k})^2$. If we let $k = \frac{W_i}{2}$, treat the upper bound as a rough approximation, and solve for t when $\mathcal{P} = 1$, we obtain $t \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$, which is roughly the expected refresh period, so $P_i(W_i) \approx \frac{1}{(2 \cdot s_i)^2} \cdot (W_i)^2$. In relation to our general expression in Section 4, for random walks $Z_i = \frac{1}{(2 \cdot s_i)^2}$ and $p = 2$.